Everything Depends on Your Hammer: A Systematic Rowhammer Attack Exploration on SPHINCS+

S. G. Shoaib Ahamed, Mrityunjay Shukla, Khushang Singla, and Sayandeep Saha
Department of Computer Science and Engineering, Indian Institute of Technology Bombay
{sgshoaibahamed, mrityunjay, khushangsingla, sayandeepsaha}@cse.iitb.ac.in

2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD) | 979-8-3315-1560-7/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICCAD66269.2025.11240843

***Abstract*—Secure implementation of Post-Quantum Cryptosystems (PQC) is becoming critical as the world migrates towards quantum-safe cryptography. Fault attacks (FA) are practical threats affecting cryptographic implementations on both embedded and native, multi-user systems. Although research on FAs targeting PQC algorithms is well-paced for embedded systems (typically with physical adversarial access), exploration of such threats for native systems (typically with remote adversarial access) requires further attention. The type of faults and, consequently, the attack algorithms differ significantly between embedded and native systems.**

In this paper, we present a systematic approach to evaluate PQC algorithms against Rowhammer – a remote, software-controlled fault injection vector in native and cloud-based systems. Unlike conventional ways in cryptography, where an attack is first developed in theory and then demonstrated on target platform(s), we take a system-specific approach by first characterizing the fault model of a target platform and then exploiting these faults to devise a suitable theoretical attack. This approach helps the attacker quickly converge to an attack optimized for the target system. Also, if exercised successfully for targets with low fault susceptibility, it finds attacks potentially affecting a large number of targets. Our approach is demonstrated for SPHINCS+, the NIST standard for stateless hash-based digital signatures. We discovered a novel Rowhammer-based forgery attack and demonstrated it for DDR3 and DDR4 DRAMs. The attacks were carried out on the standard implementation of SPHINCS+ and for the *liboqs* library. To the best of our knowledge, this is the first work proposing a forgery attack using Rowhammer on SPHINCS+.

***Index Terms*—Fault Attack, Rowhammer, Post-Quantum-Cryptography**

I. Introduction

The looming threat of quantum computers to state-of-the-art public-key cryptography [29] has compelled the world to embrace "quantum-safe" public-key algorithms – the so-called Post-Quantum-Cryptosystems (PQC) [5]. Multiple PQC algorithms, each based on a distinct "quantum-hard" class of problems, have been standardized by the National Institute of Standards and Technology (NIST) in the recent past [1]. While maintaining a portfolio with such diverse classes of algorithms is justified from the point of crypto-agility, secure and efficient implementation of such a wide variety of algorithms poses a great challenge. It is well-known that cryptographic algorithms, if not implemented with special care, may leak their secret through physical channels, such as (secret-dependent) timing of computation [17], power consumption/electromagnetic radiation [21], and maliciously induced faults [24]. Several works have emphasized the need for resiliency against such implementation-based attacks throughout the standardization process [7], [23], [26].

Among such implementation-based attacks, fault attack (FA) is the only one where the adversary actively perturbs the computation. FAs are also nontrivial to prevent, as conventional fault-tolerance measures often end up introducing new sources of leakage [28]. Moreover, fault injection has been shown feasible on a wide variety of platforms – from small embedded devices to multi-user and remotely accessible server/cloud servers [13], [24]. Therefore, evaluating PQC algorithms against this threat is extremely important, especially at this stage, while their deployments as software libraries have started taking place. However, discovering fault attacks is a nontrivial task – even for one specific algorithm, the attacks involve intricate mathematical analyses that vary depending on the physical characteristics of faults (*fault model*). Due to such complexities, exploration of attacks has largely remained a manual process to be performed separately for each algorithm. Quite evidently, this process should be repeated for each PQC class, possibly with fault models from different platforms.

In this paper, we specifically focus on fault attacks originating from Rowhammer-induced bit-flips. Rowhammer injects targeted bit-flips in DRAMs, however, with certain physical and implementation (software)-specific constraints. Such faults are not the same as faults typically occurring in embedded devices – the precision and granularity of injection time are higher for the embedded faults due to physical adversarial access. Therefore, the fault attacks from the embedded world do not directly scale with Rowhammer-induced faults. In this regard, we ask, *what would be a suitable approach for exploring attacks for PQC algorithms concerning Rowhammer-induced faults?* Given the mathematical rigor of finding attacks and the dependency of an attack on the fault model, it is a pertinent question for an attacker who wants to succeed in a reasonable time with a high probability. Conventionally, in fault attacks, an attack algorithm is derived first, assuming some theoretical model of faults (such as bit-flips at a specific variable). Next, the attack is practically evaluated on some target platform(s), establishing its feasibility. This approach has also been followed in previous works addressing Rowhammer attacks on PQC [3], [8]. However, such an approach for attack exploration has one critical limitation – it does not often comply with the varying susceptibility of DRAMs to bit-flips, which is a well-known fact in Rowhammer [10], [14], [16], [31]. A theoretically found attack may or may not be applicable on a given target, depending on the affinity of different bit locations in target's DRAM towards flips, over which the attacker has no control. From an attacker's perspective, it is always time-saving and optimal to derive attacks tailored to the target system.

We, therefore, take the following approach: *we first characterize the faults for the DRAMs we target, and then, based on the type of faults, devise theoretical attacks having a high probability of success for those platforms.* As discussed, this approach fits better with the attacker's goal. Moreover, successfully exercising this approach for DRAMs having low to medium bit-flip susceptibility (which is also the case for us) results in attacks that apply to a large set of DRAMs with varying susceptibility to flips. Showing the existence of such attacks would be an interesting outcome for attack exploration. However, doing any such exploration with proper steps is still nontrivial, given the complex dependency of Rowhammer faults on the implementation and the DRAM characteristics. In this paper, we provide templates to characterize the constraints imposed by the implementation and the DRAM, and also, through a practical

Authors acknowledge the support from the IITB TrustLab Reseacrh Grant

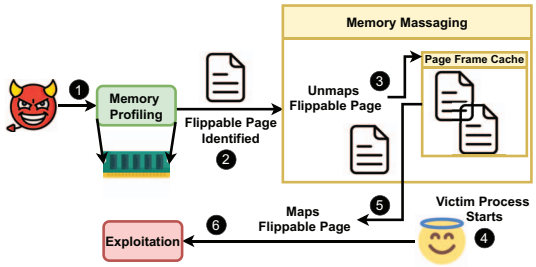

Fig. 1: Exploitation methodology for Rowhammer bit-flips

demonstration, illustrate how to meaningfully apply them. While our approach is generic and applicable to any cryptographic algorithm, it is particularly useful in the PQC context as the algorithms are diverse, new, and complex. Therefore, quickly converging to practical attacks is relatively complex. We also note that a similar philosophy has been tried to attack neural networks [33], but not for PQC algorithms, for which attacks are mathematically complex.

We demonstrate the philosophy mentioned above with SPHINCS+ [4] – the SLH-DSA (Stateless Hash-Based Digital Signature Algorithm) finalized by NIST. While the lattice-based algorithms such as Kyber [26] [7] and Dilithium [18], and multivariate signatures [19] have already received significant attention from the implementation security perspective (including Rowhammer) [3], [8], [22], [23], SPHINCS+ is relatively less explored. The only existing work addressing its implementation security is the *Grafting tree attack* [6], [11], which proposes fault-assisted forgery attacks on embedded platforms. However, to the best of our knowledge, none of the works have explored the security of SPHINCS+ or any other hash-based signature on Rowhammer attack settings (except for the one in [2], which only performs a verification bypass and not a forgery). We have successfully identified a novel attack on SPHINCS+ using Rowhammer, and practically demonstrated it for systems with DDR3 and DDR4 DRAMs, for the open-source standard implementation [9], and the implementation from the *liboqs* library [30]. Our attack corrupts the Sign function of SPHINCS+ for generating faulty signatures and, using those faulty signatures, constructs a *universal forgery*, that is, *creates valid signatures for new messages without knowing the secret-key*. The attacks were carefully crafted, taking into account the bit-flip susceptibility of the DRAMs in our possession.

The rest of the paper is organized as follows: In Sec. II, we present the background on Rowhammer and SPHINCS+. In Sec. III, we formally describe the constraints on Rowhammer faults and characterize them in the context of our targets. The new attacks on SPHINCS+ are described in Sec. IV, with the experimental details presented in Sec. V. We conclude in Sec. VI.

II. Background

A. Rowhammer

Constantly shrinking feature size of memory cells has resulted in high packaging density, and, therefore, a reduced cost-per-bit of memory in modern DRAMs. However, a "side-effect" of this small feature size is reduced reliability – activating the wordline of a DRAM row causes the capacitors in the neighboring rows to discharge due to parasitic current. A sufficient number of activations, before a DRAM refresh (typically every 64 ms), may cause the charge stored to go below a threshold, eventually causing a bit-flip in a neighboring row.

Rowhammer [16] maliciously exploits this reliability issue to induce bit-flips at memory locations which are otherwise inaccessible by the adversary from its own paged virtual address space (or, simply, *address space*). To perform bit-flips, the adversary has to access two/multiple DRAM rows repeatedly (*aggressor rows*) associated with its own address space, which are adjacent to certain rows associated with the address space of a victim (*victim rows*).

Malicious exploitation strategies for Rowhammer-induced bit-flips have been subjected to extensive research [3], [8], [13], [20], [27], [32]. In the widely accepted threat model, the attacker process ($\mathcal{A}$) and the victim process ($\mathcal{V}$), with $\mathcal{A}$ having user-level privileges, run on the same hardware, sharing the DRAM. However, they have address space isolation ensured by the OS via virtual memory. The goal of $\mathcal{A}$ is to flip specific bits in the address space of $\mathcal{V}$, which would lead to some exploitable outcome (*e.g.*, recovering secret-keys). This is achieved in steps as follows steps [20], [27] (ref. Fig. 1):

1) Memory Profiling: $\mathcal{A}$ explores the DRAM to discover flippable bit locations (or simply, *flippable bits*). This step is called *profiling stage* of the attack. The attacker: 1) allocates memory in its own address space, 2) identifies which DRAM banks correspond to these virtual addresses (either via the timing channel based on row-buffer conflict [25], or by exploiting Transparent Huge Pages aka. THPs [15]), 3) finds contiguous rows, and 4) repeatedly access some of these rows to identify the potential aggressors and victim rows[1].

2) Memory Massaging: After identifying the flippable bit locations in the profiling phase, $\mathcal{A}$ lures $\mathcal{V}$ to allocate its memory (in chunks of OS pages) in a specific manner, so that the vulnerable locations in $\mathcal{V}$'s address space (some variables, data structures or opcode) align with the flippable locations in the DRAM. This is achieved by exploiting the allocation policy of *page-frames* (fixed-size blocks in physical memory where an OS page can reside) in modern Linux-based systems. The Linux kernel maintains a per-core, last-in-first-out software cache, called Page-Frame Cache (PFC). PFC holds and returns the most recently deallocated page frame upon receiving a request from any process, irrespective of its process ID. Whenever $\mathcal{A}$ finds a *suitable* flippable bit (a bit in DRAM aligning with the target bit in $\mathcal{V}$'s page) during profiling, it releases the OS page corresponding to the potential victim row. This adds a page-frame to the PFC. While such page releasing happens in synchronization with victim's memory request, PFC can allocate the page-frame with the flippable bit to the OS page of $\mathcal{V}$, in which $\mathcal{A}$ wants the flip.

3) Hammering: After manipulating the page allocation of the victim, the attacker accesses the aggressor rows in its own address space a sufficient number of times to cause a bit-flip in the victim process. However, successful bit-flipping requires a minimum time window (typically at least one DRAM refresh cycle of 64 ms). Therefore, the victim process (more precisely, the functions which are to be exploited) must run at least for this time window. In certain cases, the victim runtime can be increased significantly with an additional attack called performance degradation [8].

B. SPHINCS+

1) Notations: In this paper, hash functions and Pseudorandom Functions (PRF) are denoted with bold capitals (e.g., **F** or **H** or **PRF**). The internal functions, signature schemes, and certain contextual data of SPHINCS+ are represented in sans-serif font. Any implementation (code) of a function is represented in `typewriter` font. $(\mathsf{sk}_1, \mathsf{sk}_2)$ and $(\mathsf{pk}_1, \mathsf{pk}_2)$ denote the secret-key and public-key of SPHINCS+, respectively. The signature generated by Sign function

[1]We note that the profiling stage should be preceded by a reverse-engineering step, in which the adversary figures out how a (physical) address maps to a particular DRAM bank. This mapping is system-specific and is a one-time effort for a particular processor family.

TABLE I: SPHINCS+

KeyGen:
1: Randomly pick $(\mathsf{sk}_1, \mathsf{sk}_2)$ where $|\mathsf{sk}_1| = |\mathsf{sk}_2| = n$ bytes.
2: Generate $(\mathsf{pk}_1, \mathsf{pk}_2)$ where:
2a: pk_1 is the public-key of SPHINCS+ hypertree.
2b: pk_2 is a public seed for hash functions.

Sign$(msg, \mathsf{sk}_1, \mathsf{sk}_2)$:
1: $R := \mathbf{PRF}_{msg}(\mathsf{sk}_2, opt)(msg)$, where opt is a random string and $\mathbf{PRF}_{msg}$ is a pseudorandom function.
2: $(\mathsf{md}, \mathsf{ADRS}) = \mathbf{H}_{msg}(\mathsf{pk}_1, R)(msg)$.
3: Sign md with the FORS at address ADRS using sk_1 producing σ_F, and let pk_F be the FORS public-key.
4: Sign pk_F with the hypertree to produce $\sigma_{HT} = (\sigma_0^X, \cdots, \sigma_{d-1}^X)$.
5: **return** $\Sigma = (R, \sigma_F, \sigma_{HT})$

Verify$(msg, (\mathsf{pk}_1, \mathsf{pk}_2), \Sigma)$:
1: $(\mathsf{md}, \mathsf{ADRS}) = \mathbf{H}_{msg}(\mathsf{pk}_1, R)(msg)$.
2: Extract pk_F, the public-key of the FORS using ADRS, md and σ_F.
3: Extract pk_{HT} the public-key of hypertree using ADRS, σ_{HT}, and pk_F.
4: **return** True if $\mathsf{pk}_{HT} = \mathsf{pk}_1$ and False, otherwise.

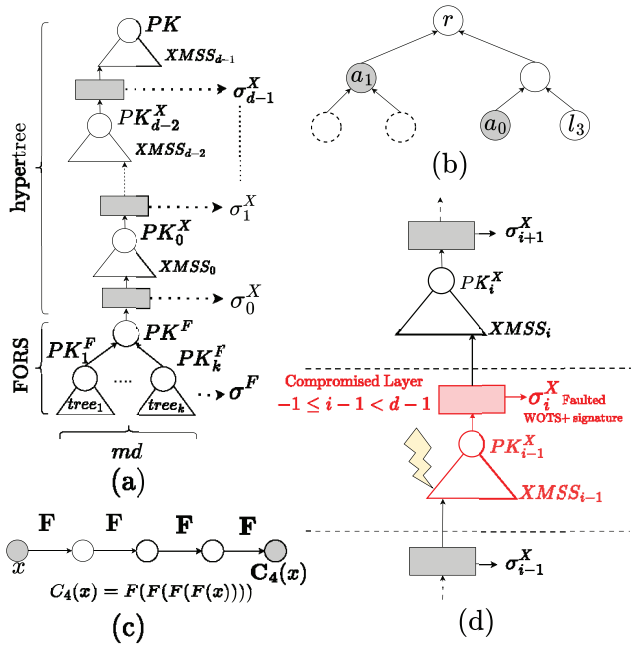

Fig. 2: SPHINCS+: (a) Hypertree, (b) Hash tree: l_3 is the leaf and (a_0, a_1) is the corresponding auth; (c) WOTS+ chaining function; (d) faulted XMSS tree for Grafting tree attack

is denoted as Σ, while (sub) signatures generated by components of the Sign are denoted as σ (with proper subscripts and superscripts).

2) General Idea: SPHINCS+ [4] is a stateless hash-based signature scheme based on the hardness of solving the decisional second preimage resistance problem in hash functions even with quantum computers. Table. I presents the pseudocode for SPHINCS+, and Fig. 2(a) depicts its internal structure. The general idea of any hash-based signature is that the public-key can be computed/extracted from the signature without having the secret-key. The verification happens by extracting the public-key from the signature and checking if it matches the stored public-key. Most of such signatures, however, can use their secret-key only once (or a few times) and, therefore, are known as one-time signatures (OTS) or Few-time signatures (FTS). OTS and FTS also need to maintain their states (and are called *stateful*) to prevent repeated use of the same secret-key. SPHINCS+ combines several such hash-based signature schemes (OTS and FTS) in a tree-like structure called *hypertree* to get rid of the statefulness condition, while still respecting the constraints, such as one-time usage of secret-keys for OTSs.

3) Hash Tree: A core component of SPHINCS+ is a *hash tree* as depicted in Fig. 2(b). A hash tree is a binary tree of height $z > 0$ which compresses 2^z leaf nodes to a single (root) node r, by hashing two nodes at a time using a hash function $\mathbf{H}$. An *authentication path* (auth) is a set of (leaf and non-leaf) nodes from this hash tree, which can reconstruct the root even without all 2^z leaf nodes. An example authentication path is depicted in Fig. 2(b) (nodes a_0, a_1) for leaf l_3. Finally, each hash tree is associated with a unique byte-string called address, which consists of a *tree-index* to uniquely identify a tree and a *leaf-index* to uniquely identify a leaf within a tree. In SPHINCS+, we interpret hash tree as a function, which, given some leaf-index j as input, returns a pair $(\mathsf{leaf}_j, \mathsf{auth}_j)$, where leaf_j is the j-th leaf and auth_j is the authentication path which, along with leaf_j, can regenerate the root of the hash tree. With slight modification, a hash tree can be used as an OTS where its root is used as the public-key and each leaf node stores a public-key secret-key pair. The public-keys of the leaves are hashed to generate the hash tree's root. Each leaf can be used to sign only one message. As an OTS, a hash tree returns $(g(\mathsf{leaf}_j), \mathsf{auth}_j)$, where the $g(\mathsf{leaf}_j)$ is the signature corresponding to the leaf_j. Verification happens by first generating the public-key corresponding to leaf_j from $g(\mathsf{leaf}_j)$ and then generating the root r of the hash-tree with the authentication path. Finally r is matched to the stored public-key of the hash tree.

4) Forest Of Random Subsets (FORS): FORS is a Few-Time Signature (FTS) scheme in which a key-pair can be used multiple times, but its security slowly declines with the number of uses. FORS generates a forest of k hash trees, and signs the message using these hash trees. The secret-key of FORS (SK^F) is a set of all the leaves of these hash trees (generated from sk_1, pk_2, and an address given as input), and the public-key (PK^F) is the hash of all the hash tree roots. For a given message digest $\mathsf{md} = (m_1||m_2, \cdots ||m_k)$ ($|m_j| = a$ bits), FORS generates a signature $\sigma^F = (\sigma_1, \sigma_2, \cdots, \sigma_k)$, where σ_j is the $(\mathsf{leaf}_{m_j}, \mathsf{auth}_{m_j})$ from the j-th ($1 \le j \le k$) hash tree given m_j as the leaf-index. The verification happens by generating the public-key of each hash tree from the signature, hashing them together, and checking against the stored FORS public-key.

5) Winternitz One-Time Signature (WOTS+): WOTS+ is a OTS used in SPHINCS+. WOTS+ uses a *chaining pseudorandom function* $\mathbf{C}_t(\mathsf{pk}_2, \mathsf{address}, x)$, realized by $t \ge 0$ consecutive applications of a hash function $\mathbf{F}$, parameterized by pk_2, a secret x, and a unique byte-string address (ref. Fig. 2(c)). In other words, $\mathbf{C}_t(\mathsf{pk}_2, \mathsf{address}, x) = \mathbf{F}(\mathbf{F}(\cdots t\ times)(\mathsf{pk}_2, \mathsf{address}, x)\cdots)$. For simplicity, we refer this function as $\mathbf{C}_t(x)$, or $\mathbf{C}_t$, whenever x is implied. The secret-key of WOTS+ is $SK^W = (s_1, s_2, \cdots s_l)$, where each $s_j = \mathbf{PRF}(\mathsf{pk}_2, \mathsf{address}, \mathsf{sk}_1)$ with $1 \le j \le l$. The public-key is $PK^W = (p_1, p_2, \cdots p_l)$, where $p_j = \mathbf{C}_{w-1}(s_j)$, with w being a public constant. Given input message $m = (b_1||b_2|| \cdots ||b_l)$, where $0 \le b_j < w$ and $|b_j| = \log w$ bits, WOTS+ returns $\sigma^W = (\sigma_1, \sigma_2, \cdots \sigma_l)$, where $\sigma_j = \mathbf{C}_{b_j}(s_j)$. For verification, it computes $p_j = \mathbf{C}_{w-1-b_j}(\sigma_j)$ for $1 \le j \le l$, and checks if it matches PK^W.

6) XMSS: XMSS is a signature scheme which combines $2^{h'}$ WOTS+ signatures using a hash tree. The main idea is to have $2^{h'}$ WOTS+ key-pairs as the leaf nodes of a hash tree, where the hash tree is applied on the public-keys of the WOTS+ key-pairs, generating an XMSS public-key PK^X. Basically, this is a hash tree-based OTS where each leaf of the hash tree is a WOTS+ key-pair. Given a message, one of the leaves (say the λ-th leaf) is selected based on a unique address to sign the message using the WOTS+ secret-

key. This generates a WOTS+ signature σ_λ^W. The XMSS signature is $\sigma^X = (\sigma_\lambda^W, \mathsf{auth}_\lambda)$, where the auth_λ is the authentication path generated from the hash tree. Each WOTS+ key-pair can be used to sign only one message. During verification, the WOTS+ public-key (PK_λ^W) is extracted from σ_λ^W of the signature. Then the XMSS public-key (root of the hash tree) is generated from PK_λ^W and auth_λ and matched to the stored XMSS public-key.

7) Hypertree: The hypertree generates a tree in which each node is an XMSS, with the XMSSs above signing the XMSSs below (ref. Fig. 2(a)). The XMSSs in the hypertree are denoted as *layers*. There are a total of d layers (counted from 0 to $d-1$), each having an XMSS of height h'. The public-key of the hypertree is the root of the topmost XMSS tree ($PK^{HT} = PK_{d-1}^X$), whereas the secret-key (SK^{HT}) is the SPHINCS+ secret-key. Each XMSS in the hypertree has a unique tree-index. *A leaf of the* XMSS *at layer* $i+1$ *signs the root of the* XMSS *at layer* i. Both tree-index and leaf-index are part of a unique byte-string address. The signature is given as $\sigma^{HT} = (\sigma_0^X, \sigma_1^X, \cdots \sigma_{d-1}^X)$, where each σ_i^X is a XMSS signature at the i-th layer generated by signing the XMSS public-key of the $(i-1)$-th layer (PK_{i-1}^X) with a WOTS+ leaf (of an XMSS) at the i-th layer. During verification, PK_{d-1}^X is matched with stored public-key.

Referring to the Table. I, and the Fig. 2(a), SPHINCS+ generates a message digest md from a message *msg* with a hash. An optional random string *opt* is also provided as input if the scheme is *randomized*. A unique byte-string ADRS is generated along with md, which is an input to each internal signature scheme (as address) deciding the tree and leaf-indices. md is first signed by the FORS. Next, the FORS public-key is signed by the hypertree, generating the final signature. The verification procedure extracts the public-keys from each underlying signature scheme and checks against pk_1.

C. The Grafting Tree Attack [6] [11]

1) Main Idea: The grafting tree fault attack performs a universal forgery on SPHINCS+ without knowing the secret-key. The fault corrupts the Sign function, making a WOTS+ key-pair sign more than one message. Here, "message" refers to an XMSS public-key. If the attacker knows any two such messages and their corresponding WOTS+ signatures, it can forge a WOTS+ signature (i.e., can generate a valid signature for a new message without knowing the WOTS+ secret-key) with probability $\sim 2^{-34}$ [12]. With more messages and signatures available, this probability rapidly grows to 1. The grafting tree attack uses this WOTS+ forgery to construct a universal forgery on SPHINCS+.

2) WOTS+ Forgery: The attack targets the WOTS+ at the $(i-1)$-th layer (mostly $i = d-1$), injecting a transient fault, only corrupting the computation of the XMSS hash tree at that layer. This generates a corrupted XMSS public-key $PK_{i-1}^{'X}$. Let the λ-th leaf from the i-th layer signs this corrupted $PK_{i-1}^{'X} = (b_1^{'}, \cdots b_l^{'})$, generating a faulty signature $\sigma_i^{'X} = (\sigma_\lambda^{'W}, \mathsf{auth}_\lambda^{'})$, where $\sigma_\lambda^{'W} = (\sigma_1^{'}, \cdots \sigma_l^{'})$. Each $\sigma_j^{'} = \mathbf{C}_{b_j^{'}}(s_j)$, with $0 \le b_j^{'} < w$. Observe that: a) for each j, there can be w possible $(b_j, \mathbf{C}_{b_j})$ pairs (where $\mathbf{C}_{b_j} = \mathbf{C}_{b_j}(s_j)$). b) Each $\mathbf{C}_{b_j}$ (resp. $\mathbf{C}_{b_j^{'}}$) is a (correct/faulty) WOTS+ signature output. c) The knowledge of $\mathbf{C}_{b_j=\alpha}$ can generate all $\mathbf{C}_{b_j>\alpha}$ without knowing corresponding secret-keys. This is because, if $b_j > \alpha$ and $\mathbf{C}_\alpha$ is known, then $\mathbf{C}_{b_j} = \mathbf{C}_{b_j-\alpha}(\mathbf{C}_\alpha)$. Based on these observations, the adversary tries to collect as many $(b_j, \mathbf{C}_{b_j})$ pairs as possible, one generated from the correct computation (denoted b_j), and the others from the faulted $PK_{i-1}^{'X}$s (denoted $b_j^{'}$). The goal is to obtain b_js (or $b_j^{'}$s) close to 0 for each j. Once the adversary obtains such "low" values, it can forge a signature exploiting property c). More precisely, for a new $PK_{new}^X = (b_1^{new} || b_2^{new}, \cdots b_l^{new})$, the adversary can find $\sigma_{j,new} = \mathbf{C}_{b_j^{new}}$ using the known $(b_j, \mathbf{C}_{b_j})$ and $(b_j^{'}, \mathbf{C}_{b_j^{'}})$ pairs. However, it is worth mentioning that forging is possible even with one correct and one faulty $(b_j, \mathbf{C}_{b_j})$ for a randomly chosen new message with probability 2^{-34}. Therefore, getting more $(b_j, \mathbf{C}_{b_j})$ at the cost of more faults is only a choice and not a mandate.

3) Tree Grafting: Let us define $\theta_1, \cdots \theta_l$ as the lowest values for each b_j (resp. $b_j^{'}$) obtained by the adversary, along with the corresponding $\mathbf{C}_{\theta_j}$s. The fault has been injected at the $(i-1)$-th layer during a signature computation. At first, the adversary randomly chooses $(\mathsf{sk}_1^*, \mathsf{sk}_2^*)$. Next, for a given message m^*, for which a forged signature is to be generated, the adversary performs the following: 1) it iterates over randomly chosen values of R (ref. line 2; Sign function; Table. I) denoted as R_{pr}, and with $(\mathsf{sk}_1^*, \mathsf{sk}_2^*)$, partially computes the signature till the $(i-1)$-th layer. Let the partially computed signature be $\Sigma_{pr} = (R_{pr}, \sigma_{pr}^F, \sigma_{0,pr}^X, \cdots \sigma_{i-1,pr}^X)$. The adversary chooses a Σ_{pr} for which the tree-index at layer $i-1$ matches with the tree-index of the faulted XMSS tree at the same layer. 2) For this Σ_{pr}, the adversary also computes the $PK_{i-1,pr}^X = (r_1, \cdots r_l)$, and checks if $r_j \ge \theta_j$, $\forall j$. If both constraints on the tree-index and r_js are satisfied, the adversary computes a $\sigma_i^{*X} = \langle(\mathbf{C}_{r_1-\theta_1}(\mathbf{C}_{\theta_1}), \cdots, \mathbf{C}_{r_l-\theta_l}(\mathbf{C}_{\theta_l})), \mathsf{auth}\rangle$, which is a forged XMSS signature for $PK_{i-1,pr}^X$. The corresponding forged XMSS tree at $(i-1)$-th layer is called the *grafted subtree*. 3) From a correct signature, the adversary collects $(\sigma_{i+1}^X, \cdots, \sigma_{d-1}^X)$ which eventually leads to the correct public-key pk_1. Finally, it returns the forged signature as: $\Sigma_{pr} = (R_{pr}, \sigma_{pr}^F, \sigma_{0,pr}^X, \cdots \sigma_{i-1,pr}^X, \sigma_i^{*X}, \sigma_{i+1}^X, \cdots, \sigma_{d-1}^X)$.

III. Systematizing Rowhammer Faults

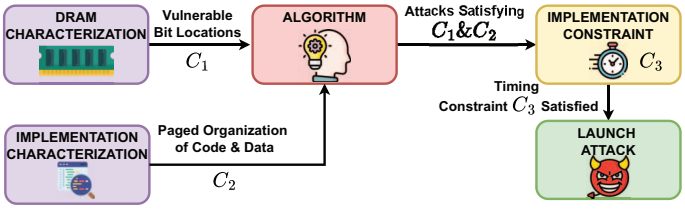

Fig. 3: End-to-End Attack Exploration Algorithm

As already pointed out in the introduction, in this paper, we first characterize the fault model of a target platform and, based on that, craft the theoretical attack. For an attacker who wants to attack a specific system and a crypto-implementation running on it, it is much easier to rely on faults that are happening with high probability for that specific system, rather than forcing the realization of an already existing theoretical fault model on that system. From a white-hat perspective, the vulnerability of a given implementation, while deployed on a specific target system, can be judged. In addition, if an attack is found that occurs on multiple DRAMs with relatively low susceptibility to flips, it can be considered highly severe.

Fault attacks are artifacts of: 1) mathematical characteristics of the target cryptographic algorithm, 2) the implementation of the algorithm, and 3) the physical characteristics of the device being attacked. In order to assist the attacker, we therefore present templates for capturing the constraints from the DRAM (*physical constraints*) and those from the implementation side (*implementation constraints*).

A. Consolidating Physical and Implementation Constraints

1) Physical Constraints: As described in Sec. II-A, Rowhammer-induced bit-flips can be directed to corrupt a particular OS page of

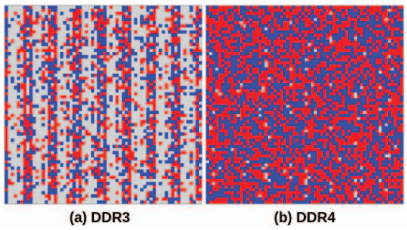

Fig. 4: Distribution of vulnerable byte offsets in a 4KB page ■ (Flip Direction $1 \rightarrow 0$) ■ (Flip Direction $0 \rightarrow 1$)

the victim in a controlled manner (memory massaging). However, bit-flips at different bit offsets within the page are physical artifacts and cannot be controlled. For a successful attack, some of the physical bit-flips must coincide with the bit that the attacker wants to flip in an implementation. Not every DRAM is equally susceptible to Rowhammer-induced bit-flips. Several works have addressed this point [10], [13], [14], [25], [27]. The second physically-imposed constraint is the polarity of the flips ($0 \rightarrow 1$ or $1 \rightarrow 0$) at a specific bit-offset. We capture such physical constraints as:

$$C_1 : \{(fl_j, pol_j)\}_{j=0}^{Q} \tag{1}$$

where $fl_j = 1$ denotes that bit at j-th bit offset within a page is flippable ($fl_j = 0$, otherwise), and pol_j denote the polarity of the flip ($pol_j = 0 \implies (0 \rightarrow 1)$ and $pol_j = 1 \implies (1 \rightarrow 0)$). Q is the number of bits within a page. For Linux-based systems, $Q = 32768$.

To illustrate how the aforementioned physical constraint behaves, we present two example bit-flip profiles for DDR3 and DDR4 DRAM modules in Fig. 4. Our profiling approach is the same as mentioned in Sec. II-A in the profiling stage of the attack. For the sake of visibility, we present the byte offsets having bit-flips in the figures, with the polarities denoted with different colors. The exact bit-flip statistics are presented in Table II, where the number of distinct flippable bit offsets within a page has been reported. It can be observed that the DDR4 DRAM module has a lot more vulnerable byte offsets compared to the DDR3 one. However, the distribution of bit-flips in terms of flip direction is almost uniform for both cases. Since both the bit-offset and polarity must match for a successful flip, the DDR3 module, in our case, can be said to have low susceptibility to flips. It is important to note that for one particular allocation of a page-frame, we hardly observed more than one bit-flip. The implication is that *although several bit offsets within a page can be flipped, we can observe (with high probability) at most one bit-flip per OS page at a time.*

TABLE II: Memory Profiles of our two attack configurations

Module	Total bit-flips	Direction $0 \rightarrow 1$	Direction $1 \rightarrow 0$
DDR3	4129	2124	2005
DDR4	17770	8818	8952

2) Implementation Constraints: The second type of constraints originates from the target's implementation. Any software implementation is a collection of constants, static and dynamic variables, data structures, and opcodes, organized in chunks of OS pages. This page-wise organization should be treated as a constraint for attack, especially given the fact that one bit can be flipped within an OS page at a time (ref. last paragraph). We interpret the entire implementation as a set of pages $\{Pg_j\}_{j=1}^{\#pg}$, where $\#pg$ denotes the total number of pages in the implementation. We emphasize that the Pg_j refers to the same object (OS page) for which we imposed the constraint C_1. Each page has a type typ from the set $\mathsf{T} = \{code, data\}$, where the $data$ refers to pages containing data elements, such as variables and data structures, and $code$ refers to pages holding the opcodes[2]. Furthermore, to formally specify the number of flips possible on a page for a particular page-frame allocation, we use an integer $\#fnum$. As noted in the previous section, $\#fnum_j \leq 1$ for most practical cases. The page-level constraint can be defined as follows:

$$C_2 : \{\langle Pg_j, typ_j, \#fnum_j \rangle \mid \#fnum_j \leq 1\}_{j=1}^{\#pg} \tag{2}$$

Whether a fault injection would succeed or fail also depends on a third factor: *how long the target bit resides in the DRAM without being accessed by the processor.* The target bit is to be decided by the attack algorithm based on the constraints C_1 and C_2. The *hammering* step of the Rowhammer usually requires a finite number of accesses to the aggressor row to induce the flip, which incurs this timing-based constraint. For a successful flip to take place, the target bit must remain in the memory for at least one DRAM refresh cycle, which is typically 64/32 ms for most of the commodity DRAMs. We denote this timing threshold as Th. This constraint is represented as:

$$C_3 : \mathsf{Time}(v) > Th, \tag{3}$$

where v is the target bit. Time specifies the time interval between two consecutive accesses of v, while it resides in DRAM (and not in the caches). However, we underscore that, unlike C_1 and C_2, both of which can be applied by the attacker before or during devising the attack in theory, C_3 can only be applied after the target bit has been decided. This is because C_3 is strongly dependent on the code semantics, and checking it a priori, even before knowing which function/block of the code to be targeted, will be an overkill. However, C_3, being a timing constraint on a page, can be checked for potentially exploitable bits even before flipping those bits. Therefore, it can be used to fine-tune the attacks before launching.

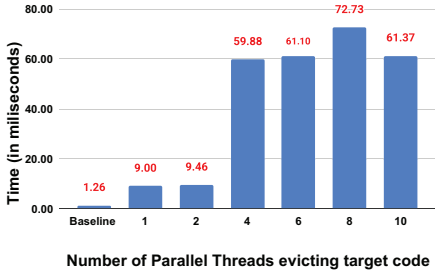

Fig. 5: Performance degradation for matrix multiplication.

We note that the timing for a bit to remain in DRAM can be artificially increased using performance degradation attacks [8]. Performance degradation has been shown instrumental for attacking PQC implementations [3]. We illustrate such performance degradation on a simple matrix multiplication program multiplying two 21×21 matrices of doubles (ref. Fig. 5). The main strategy for slowdown is to aggressively evict the most frequently used instructions of the function to be slowed down from the cache. We used a combination of three known techniques – evicting frequently used instructions from the cache using `clflush`, thrashing the last-level cache (LLC) from multiple threads, and running busy loops on the same physical and logical cores where the victim process runs [3]. As can be observed from Fig. 5, the performance degradation can lead to orders of magnitude slowdown – from 1.26 milliseconds to 72.73 milliseconds, by increasing the number of threads. Unfortunately, such slowdown can only be achieved up to a certain level, strongly depending on the code semantics. Aggressive use of the three aforementioned tricks

[2]Note that, while this typing of pages is not as detailed as in Linux memory layout it contains the information we need for attack

may also lead to a reduced slowdown (ref. Fig. 5), and may also slow down the hammering process, hindering bit-flipping. Depending on the code semantics, the slowdown may not be effective, as will be shown in our results in the next section.

The systematic flow for attack identification is depicted in Fig. 3. As mentioned, the physical and implementation constraints are supplied to the attacker, who works out the attack in theory. The attacker first identifies bit-flips that are exploitable and then checks if they comply with C_1 and C_2. Certain constraints, such as C_3, are checked after finding attacks theoretically.

IV. Exploration of Attacks on SPHINCS+

Equipped with the Rowhammer-induced constraints, we now elaborate on how to exploit the faults happening in a given DRAM for targeting SPHINCS+.

A. The SPHINCS+ Implementation

Multiple libraries have implemented SPHINCS+ recently. In this section, we mainly look into the standard implementation of SPHINCS+ available in [9]. Later, we also provide results on SPHINCS+ implementation from the *liboqs* library [30]. The three APIs for SPHINCS+ are `KeyGen`, `Sign`, and `Verify`. In this paper, we keep ourselves limited to the `Sign` function. The functions called by the `Sign` function are as follows:

- `gen_message_random` generates R (line 1. in Sign).
- `hash_message` generates the message digest md and the address ADRS (line 2. in Sign).
- `fors_sign` is used to generate the FORS signature.
- `merkle_sign` is the function that is used to produce the XMSS signature at each layer. This function runs in a loop, implements the SPHINCS+ hypertree and WOTS+ signatures as described in Sec. II-B. Internally, this function calls a function named `treehash`, which implements the XMSS hash tree, and outputs the authentication path as a part of the final signature. The complete call graph is depicted in Fig. 6.

The implementation of SPHINCS+ does not contain any constant table. The only constants used in the code are associated with the size and structure of the internal signature schemes (FORS, WOTS+, XMSS, and hypertree). The entire construction uses fixed-sized arrays to hold the inputs and outputs of hash function calls.

Attack Intuition: Given the implementation style, we could quickly rule out the possibility of certain attacks. In particular, no attack could be found by faulting constant data and tables. We note that such attacks have been described before for table-based implementations of block ciphers such as AES [34]. Also, observing the rarity of bit-flips in the DDR3 DRAM, we aim to keep our attack flexible enough with respect to bit-flip locations and the number of bit-flips. *Ideally, we want an attack to succeed with a single flip happening at any bit offset (or, at least, happening among a large number of candidate offsets compliant with C_1) within a page.*

B. Attack 1: The Grafting Tree

The first attack that is potentially feasible is the grafting tree attack described in Sec. II-C. This is because the grafting tree attack requires a random corruption of a particular XMSS at a target layer i, which can be achieved by faulting any bit within the `merkle_sign` function implementing the XMSS. However, the fault must happen at a specific time at a particular call to the `merkle_sign` or any of its callee functions. In this regard, we observed that `treehash` is the function in which such a fault can be made. More specifically, we need to fault the data (which is changing dynamically between different calls) being processed within `treehash` at a particular call.

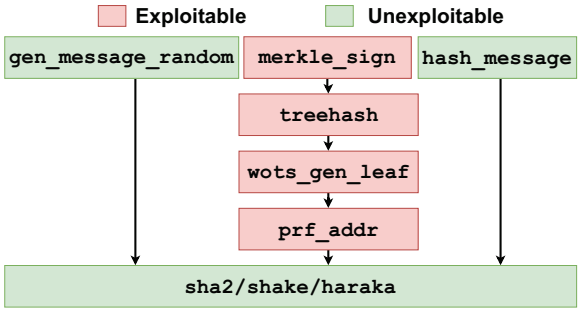

Fig. 6: Call-graph of Sign . Exploitable functions are in red.

1) Results of Rowhammer: C_1 and C_2 were easily satisfied for both the DDR3 and DDR4 DRAMs, as fault at any bit works. However, the fault has to be made at a specific call to a (frequently called) function. Due to frequent calls, the function, its code, and the data it processes mostly remain in the cache, making C_3 hard to satisfy without performance degradation. Unfortunately, we found C_3 being violated for all potential bit locations, even with performance degradation. More specifically, with performance degradation, we could slow down the function `treehash` from 0.05 to 2.24 ms, which was still insufficient to satisfy C_3. The failure of the attack signifies the need to consider the constraints while deriving an attack.

C. Attack 2: Novel Persistent Fault Attack

We next target the opcodes. Opcode flipping has been shown fatal on multiple occasions in the past [10], [13], [14]. In such attacks, an opcode is changed to a different yet valid opcode by flipping a bit, which changes the semantics of the code (ref. Fig. 7(b)). Opcode-flip attack, however, affects all invocations of a corrupted function within an execution. In other words, the fault remains *persistent* on a function throughout the execution, and may corrupt the execution more than intended. In the context of PQC algorithms, a recent work utilizes such opcode flipping for attack [3]. However, this work targets a function that is called only once in an execution, and therefore, has limited corruption. In the context of SPHINCS+, we observed that functions involved in computing the R, md, or ADRS are called only once, but we did not discover any potential theoretical attack involving these functions. Potentially exploitable functions (those involved with WOTS+, XMSS, or FORS computation) are called several times, and corrupting any one of them at the opcode level spills the fault in the entire execution. We now present an attack, which exploits such excessive corruption.

1) The Mathematical Intuition: Let us recall: 1) (Message-dependent) ADRS uniquely determines the tree-index and leaf-indices of every XMSS tree present in a signature. In other words, for every message, the hypertree passes through a unique path constructed with XMSSs determined by ADRS. 2) WOTS+ key-pairs construct the leaves of XMSSs. For a given secret-key $(\mathsf{sk}_1, \mathsf{sk}_2)$, WOTS+ key-pairs are uniquely determined by the tree-index and leaf-indices of the corresponding XMSS tree. 3) An XMSS signature at i-th layer (σ_i^X) is computed by signing the public-key of the XMSS tree from the $(i-1)$-th layer (PK_{i-1}^X), with a WOTS+ key-pair from the i-th layer at a specific leaf-index. An XMSS signature $\sigma^X = (\sigma_\lambda^W, \mathsf{auth}_\lambda)$ contains a WOTS+ signature σ_λ^W at leaf λ and corresponding auth_λ.

Observation 1. *Consider a fault f_{i-1} at the XMSS tree at $(i-1)$-th layer resulting in a faulty public-key $PK'^X_{(i-1)}$. Let the fault only corrupt the hash tree computation of the XMSS. The fault affects σ_i^X, but it does not affect the XMSS public-key at the i-th layer (PK_i^X). Therefore, while σ_i^X will be faulty, PK_i^X will be fault-free.*

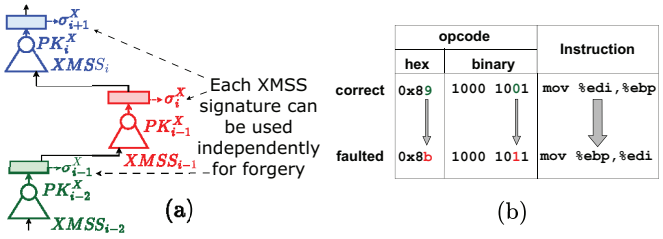

Fig. 7: (a) Effect of persistent fault, (b) Single bit-flip in an opcode.

Observation 1 is an artifact of WOTS+ and XMSS signing processes. At each layer, the XMSS hash tree is computed over $2^{h'}$ WOTS+ public-keys as leaves. More specifically, at the i-th layer, each λ-th WOTS+ public-key $PK_\lambda^W = (p_1, p_2, \cdots p_l)$ is hashed to a single value $\mathbf{T}(PK_\lambda^W)$, and $\{\mathbf{T}(PK_\lambda^W)\}_{\lambda=1}^{2^{h'}}$ are processed by the hash tree to create the XMSS public-key PK_i^X. Note that $p_j = \mathbf{C}_{w-1}(s_j)$ for $1 \le j \le l$, that is, input to $\mathbf{C}$ does not depend on the $PK_{i-1}^{\prime X}$, which is faulted due to the fault at layer $i-1$. *Therefore, the* XMSS *public-key at i-th layer does not depend on the (possibly faulty) public-key of the $(i-1)$-th layer. But the* XMSS *signature at i-th layer depends on the (possibly faulty) public-key at $(i-1)$-th layer.*

Observation 2. *Consider multiple faults $\{f_i\}_{i=0}^{d-1}$ with f_i corrupting the i-th layer of* hypertree*, in a way so that only the hash trees are corrupted at each layer. We get a faulty* hypertree *signature $\sigma'^{HT} = (\sigma_0'^X, \cdots \sigma_{d-1}'^X)$, where each $\sigma_i'^X$ only depend on f_{i-1}.*

Observation 2, which is caused by Observation 1, is illustrated in Fig. 7(a). *Observation 2 implies that: even if all the* XMSS *trees are faulted, each faulted* XMSS *tree (and associated* WOTS+ *signatures) can be exploited independently for attack.* Now we recall the grafting tree attack – it performs the WOTS+ forgery based on only one faulty XMSS signature $\sigma_i'^X$ (possibly with many faulty values for this $\sigma_i'^X$). Since, as per Observation 2, this signature is independent of all other faulty signatures and faults, it can be used independently for creating a WOTS+ forgery. *Therefore, we can treat the multiple fault scenario equivalent to a single fault scenario, and can apply the same tree grafting as before, even if all the* XMSS *trees are faulty.* Such a situation happens when we fault the opcode of the `treehash` function and some other related functions (ref. Table. III).

2) The Attack Algorithm: The attack proceeds as follows:

(i) The adversary collects some correct signatures for some randomly chosen messages.
(ii) It induces the fault at a chosen opcode and collects several faulty signatures.
(iii) From the faulty signatures it chooses a layer i and performs WOTS+ forgery as described in the grafting tree attack.
(iv) Finally, it performs the tree grafting and returns a forged signature for a given message msg^*.

3) Deterministic Vs. Randomized Signatures: In the simplest case of deterministic signatures, the attacker generates one correct and one faulty signature for a message msg. Since the ADRS does not get faulted, the tree and leaf indices of the correct and faulty signatures remain the same, giving one correct and one faulty set of $(b_j, \mathbf{C}_{b_j})$ pairs for a WOTS+ leaf. With a single correct and faulty signature pair, the attacker performs the tree grafting with roughly an exhaustive search of 2^{34} (ref. Sec. II-C).

As it can be observed in Table. I, in the randomized implementation of SPHINCS+, the message digest md and the address ADRS depend on a randomly chosen string opt. Consequently, the path chosen (i.e., tree and leaf indices) by the hypertree is different for each call to Sign, even if the message remains the same. Consequently, paths for correct and faulty signatures will be different.

Due to this randomization, it becomes difficult to make the same WOTS+ leaf sign a correct and a faulty XMSS public-key, as the tree and the leaf indices change at each execution. In order to attack, we exploit the fact that the number of possible XMSS trees is small while the layer index i is close to $d-1$. There is only one possible tree at layer $d-1$ with $2^{h'}$ leaves. Without loss of generality, we exploit the XMSS at $(d-2)$-th layer for grafting. There can be $2^{h'}$ such XMSSs (with $h' = 3/4/8/9$), each of which corresponds to a leaf of the XMSS tree at layer $(d-1)$. If we generate several faulty signatures, each of the signatures will pass through one of these $2^{h'}$ leaves giving $(b_j', \mathbf{C}_{b_j'})$ $(1 \le j \le l)$ pairs for each leaf. While forging, the path of the partially computed signature Σ_{pr} may pass through any of these leaves. Since $(b_j, \mathbf{C}_{b_j})$ pairs have been collected for all, the grafting can be performed at any of the leaves.

4) Results of Rowhammering: We note that the constraint C_3 is satisfied by default for this attack as flipping the opcode takes place before the victim starts executing. However, it might seem that satisfying C_1 and C_2 is harder, because one has to fault a specific bit within an opcode to make the attack work. However, it is not the case, as there are several opcodes of this kind where flipping a bit results in a successful attack. A search using our in-house GDB-based tool reveals several such fault locations across different functions as presented in Table. III. We note this list is not exhaustive. Faulting at least one of these locations results in a successful attack. Moreover, for a single injection, several faulty signatures can be collected, as the fault is persistent. This feature becomes handy for attacking randomized signatures for which several faulty signatures are needed.

TABLE III: Exploitable fault locations in standard and *liboqs* implementation of SPHINCS+ using GDB

Function	Standard implementation	*liboqs* implementation
`merkle_sign`	197	196
`treehash`	372	334
`wots_gen_leaf`	249	818
`prf_addr`	34	983

5) Exploiting Multiple Fault Locations: We observed that *each (opcode) fault location results in one set of faulty $(b_j', \mathbf{C}_{b_j'})$ $(1 \le j \le l)$ values per* WOTS+ *leaf (key-pair).* While, along with a correct $(b_j, \mathbf{C}_{b_j})$, it is sufficient for the attack, it can be improved. Faulting at different opcodes increases the number of XMSS public keys signed by each WOTS+ key-pair. This feature can significantly reduce the complexity of the tree grafting stage in our attack (ref. Sec. II-C). More precisely, we need to collect faulty signatures corresponding to different opcode flips (one flip at a time), so that each WOTS+ eventually ends up signing more than two XMSS public keys. With 3-4 (correct/faulty) such signings, the grafting happens within a few minutes. *To summarize, multiple signatures generated with one fault location cover different $(2^{h'})$* WOTS+ *leaves. However, only one faulty $(b_j', \mathbf{C}_{b_j'})$ is obtained per leaf. With multiple fault locations, we can increase the number of faulty $(b_j', \mathbf{C}_{b_j'})$ per leaf.*

V. Practical Evaluation

A. The Attack Setup

TABLE IV: Summary of the experimental setup

CPU	DRAM	Profiler
i7-4790	DDR3 8GiB	Hammertime [31]
i7-8700	DDR4 8GiB	Blacksmith [14]

1) Finding out Potential Fault Locations: One of the major steps of our systematic flow is to find out potential fault locations in the target implementation. We use an in-house GDB-based fault simulator for opcode and data fault simulation, as mentioned in the last section. It is worth mentioning that the simulator takes C_1 and C_2 into account while simulating the faults. C_3 is checked separately if needed.

2) Setup for Rowhammering: Our setup for Rowhammer attacks is presented in Table IV. The attack flow is described in Sec. II-A, and depicted in Fig. 1. Note that for finding C_1, the attacker performs an initial profiling. The profiling is repeated once again at the online phase of the attack to find the flips and corresponding OS pages. Since our attacks already take C_1 into account, we are almost certain about rediscovering at least some of the flips included in C_1.

We use modified versions of open-source HammerTime [31] (DDR3) and Blacksmith [14] (DDR4) DRAM bit-flip profilers for profiling. The HammerTime [31] uses the `pagemap` function of Linux (requiring `sudo` privilege) to find out the DRAM banks corresponding to a virtual address. There exist multiple options to get rid of `pagemap`, such as by exploiting THPs [15], or by exploiting bank conflicts through a timing channel [25]. Any of these options would make the profiling stage free of `sudo` privilege. The choice of HammerTime is entirely motivated by the ease of reusing existing stable tools. Our DDR4 setup, however, runs without any privileged (`sudo`) access. In this case, we perform the profiling on 2 MB THPs [15], for which the 21 least significant bits of the physical address and virtual address are the same. The DRAM banks and contiguous rows can be easily identified using that information [15].

After profiling, we perform memory massaging. For the (unsuccessful) grafting tree attack, the memory massaging is the same as mentioned in Sec. II-A, except that it is combined with performance degradation. However, for opcode flipping, the memory massaging differs slightly from the standard description in Sec. II-A. In this case, we have to flip a bit in the (read-only) code pages of a process, which reside in the OS *Page Cache*. Prior to victim execution, the adversary maps the binary (or shared/static library) in its own address space (using `mmap` with `PROT_READ` flag), placing the target code page at the flippable location, and generates a flip by hammering. When the victim process runs the binary, it loads the code pages again. However, the OS performs *de-duplication* of identical pages in memory, giving the victim process access to the corrupted code pages mapped by the attacker. In a nutshell, the victim ends up running a corrupted binary. To place the target code page from Page Cache to a flippable location (in the attacker's address space), the attacker performs the memory massaging using the PFC [20]. However, there can be scenarios where the victim is already running before the attack process starts. In this case, the code pages are already present in the Page Cache, and to lure these pages to a flippable location, the attacker has to relocate them in memory. This is achieved by first evicting the pages from the Page Cache using the *vmtouch* utility in Linux, and then reloading the pages from the disk [15].

The final complexity in memory massaging comes from the usage of huge pages. For the DDR4 example, we use 2MB THPs. Note that this is a choice made by the attacker, and victim pages are still of 4KB. Also, THPs never get stored in the PFC. We, therefore, used the technique proposed in [15]. In short, we invoke the `madvise` system call with the `MADV_PAGEOUT` flag to split a 2MB huge page into 512 individual 4KB pages, enabling memory massaging.

B. Hammering Open-source Implementations

We evaluate the proposed attacks on two open-source implementations – the standard implementation of SPHINCS+ [9] and the implementation from the *liboqs* library. There are differences between these two implementations in terms of their C code – especially *liboqs* uses functions having `avx` instructions (with suffix `x8`) by default, while the standard implementation uses functions without `avx` (with suffix `x1`) by default. The SPHINCS+ comes with different parameter configurations. Our proposed attacks work identically for all the configurations. However, we only report results for the `SPHINCS+-256f` variant with the SHA2 hash. The exact configuration is: security parameter (n) = 32 bytes; height of an XMSS (h') = 4; number of layers (d) = 17; number of trees in FORS (k) = 35; the WOTS+ constant (w) = 16. The number of different exploitable opcode faults is reported in Table III. For certain functions, *liboqs* has significantly more vulnerable opcodes compared to the standard implementation. However, in both cases, the number of exploitable locations is sufficient to perform the attack. For `wots_gen_leaf` and `prf_addr`, having exploitable faults is unexpected, as they do not comply with the fault locations in our theoretical attack. Thorough investigation revealed that such faults actually corrupt the contents of the caller function `treehash` while saving and restoring the context from the stack during function call. Therefore, we rather consider them as faults in the `treehash`.

TABLE V: Relation between the $\#PK^X_{i-1}$ signed by single WOTS+ key and the forging time with a timeout of 2 hours

$\#PK^X_{i-1}$ signed by single WOTS+ key	Min time (in sec)	Avg time (in sec)	Max time (in sec)	Percentage of time outs
2	1.69	3616.66	Timeout	50.00
3	1.33	418.36	3940.96	0.00
4	1.08	84.72	491.93	0.00
5	0.99	33.95	145.00	0.00
6	0.96	2.74	8.30	0.00

As pointed out in Sec. IV-C, having multiple messages per WOTS+ key-pair drastically improves the attack complexity and is quite feasible in our case due to the availability of several potential fault locations. To enable faulting of different bits, we evict (corrupted) code pages from the Page Cache using *vmtouch*, and relocate the page to a different flippable memory location. For each faulted opcode, we collect 50 signatures before performing the relocation (to accommodate attacks for randomized signatures). Each fault location results in one new message signed by a specific WOTS+ key-pair. As can be observed in Table V, the signature generation succeeds in a few seconds, even if 2 faulty signatures (along with a correct signature) are available. In our setups, finding out 2 such faulty signatures requires 4-6 hours on average.

VI. Conclusion

In this work, we present the first Rowhammer-based forgery attack on SPHINCS+ – the SL-DSA standard by NIST. The attack is derived through a systematic flow, ensuring its success even before running it on the target system, which is the most practical way for an attacker. An immediate future direction would be to explore potential countermeasures for this attack. It has already been pointed out in [11] that the only way of preventing grafting tree attacks on SPHINCS+ is to perform the computation twice and check at the end. This also holds true for the persistent fault attack we propose. However, special care has to be taken for implementing such redundancy. Calling the same corrupted function multiple times would result in a faulty signature being returned, which can be the case if OS de-duplication takes place between the redundant executions. Therefore, we suggest using hardware implementations to safeguard against Rowhammer, before suitable mathematical countermeasures are found[3].

[3]https://github.com/shoaibubaid/everything_depends_on_your_hammer.git

REFERENCES

[1] National institute of standards and technology (nist). 2022. post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization, 2022.

[2] Andrew Adiletta, M Caner Tol, Kemal Derya, Berk Sunar, and Saad Islam. Leapfrog: The rowhammer instruction skip attack. *arXiv preprint arXiv:2404.07878*, 2024.

[3] Samy Amer, Yingchen Wang, Hunter Kippen, Thinh Dang, Daniel Genkin, Andrew Kwong, Alexander Nelson, and Arkady Yerukhimovich. PQ-Hammer: End-to-end Key Recovery Attacks on Post-Quantum Cryptography Using Rowhammer . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 48–48, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[4] Daniel J Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The sphincs+ signature framework. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 2129–2146, 2019.

[5] D.J. Bernstein, J. Buchmann, and E. Dahmen. *Post-Quantum Cryptography*. Springer Berlin Heidelberg, 2009.

[6] Laurent Castelnovi, Ange Martinelli, and Thomas Prest. Grafting trees: a fault attack against the SPHINCS framework. Cryptology ePrint Archive, Paper 2018/102, 2018.

[7] Jeroen Delvaux. Roulette: A diverse family of feasible fault attacks on masked kyber. *Cryptology ePrint Archive*, 2021.

[8] Michael Fahr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray A. Perlner, Arkady Yerukhimovich, and Daniel Apon. When frodo flips: End-to-end key recovery on frodokem via rowhammer. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 979–993. ACM, 2022.

[9] Scott Fluhrer, Stefan Kölbl, Ruben Niederhagen, Joost Rijnevald, Peter Schwabe, Bas Westerbaan, and Thom Wiggers. The sphincs+ reference code, accompanying the submission to nist's post-quantum cryptography project, 2022.

[10] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Trrespass: Exploiting the many sides of target row refresh. *CoRR*, abs/2004.01807, 2020.

[11] Aymeric Genêt. On protecting sphincs+ against fault attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 80–114, 2023.

[12] Leon Groot Bruinderink and Andreas Hülsing. "oops, i did it again"–security of one-time signatures under two-message attacks. In *International Conference on Selected Areas in Cryptography*, pages 299–322. Springer, 2017.

[13] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261. IEEE, 2018.

[14] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. BLACKSMITH: Scalable Rowhammering in the Frequency Domain, 2022-05. https://github.com/comsec-group/blacksmith.

[15] Ingab Kang, Walter Wang, Jason Kim, Stephan van Schaik, Youssef Tobah, Daniel Genkin, Andrew Kwong, and Yuval Yarom. SledgeHammer: Amplifying rowhammer via bank-level parallelism. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1597–1614, Philadelphia, PA, August 2024. USENIX Association.

[16] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 361–372. IEEE Computer Society, 2014.

[17] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer, 1996.

[18] Elisabeth Krahmer, Peter Pessl, Georg Land, and Tim Güneysu. Correction fault attacks on randomized crystals-dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):174–199, 2024.

[19] Juliane Krämer and Mirjam Loiero. Fault attacks on uov and rainbow. In *Constructive Side-Channel Analysis and Secure Design: 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3–5, 2019, Proceedings 10*, pages 193–214. Springer, 2019.

[20] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[21] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[22] Puja Mondal, Suparna Kundu, Sarani Bhattacharya, Angshuman Karmakar, and Ingrid Verbauwhede. A practical key-recovery attack on LWE-based key- encapsulation mechanism schemes using Rowhammer. 22nd International Conference on Applied Cryptography and Network Security (ACNS), Abu Dhabi, UAE, 5-8 March, 2024, 2024.

[23] Koksal Mus, Saad Islam, and Berk Sunar. Quantumhammer: a practical hybrid attack on the luov signature scheme. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1071–1084, 2020.

[24] Sikhar Patranabis and Debdeep Mukhopadhyay. *Fault tolerant architectures for cryptography and hardware security*. Springer, 2018.

[25] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: exploiting DRAM addressing for cross-cpu attacks. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 565–581. USENIX Association, 2016.

[26] Prasanna Ravi, Anupam Chattopadhyay, Jan Pieter D'Anvers, and Anubhab Baksi. Side-channel and fault-injection attacks over lattice-based post-quantum schemes (kyber, dilithium): Survey and new results. *ACM Transactions on Embedded Computing Systems*, 23(2):1–54, 2024.

[27] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1–18, 2016.

[28] Sayandeep Saha, Arnab Bag, Dirmanto Jap, Debdeep Mukhopadhyay, and Shivam Bhasin. Divided we stand, united we fall: Security analysis of some sca+ sifa countermeasures against sca-enhanced fault template attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 62–94. Springer, 2021.

[29] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. IEEE, 1994.

[30] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. In *International Conference on Selected Areas in Cryptography*, pages 14–37. Springer, 2016.

[31] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*, pages 47–66. Springer, 2018.

[32] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In Haryadi S. Gunawi and Benjamin C. Reed, editors, *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 213–226. USENIX Association, 2018.

[33] M Caner Tol, Saad Islam, Andrew J Adiletta, Berk Sunar, and Ziming Zhang. Don't knock! rowhammer at the backdoor of dnn models. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 109–122. IEEE, 2023.

[34] Fan Zhang, Xiaoxuan Lou, Xinjie Zhao, Shivam Bhasin, Wei He, Ruyi Ding, Samiya Qureshi, and Kui Ren. Persistent fault analysis on block ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 150–172, 2018.